AutomaDeD: Scalable Root Cause Analysis

ParaDyn/DynInst Week
March 26, 2012
AutomaDeD (DSN ’10): Fault detection & diagnosis in MPI applications

- Collect traces using MPI wrappers
  - Before and after call

```c
MPI_Send(...) {
  tracesBeforeCall();
  PMPI_Send(...);
  tracesAfterCall();
  ...
}
```

- Collect call-stack info
- Time measurements

Find offline:
- phase
- task
- code region
Modeling Timing and Control-Flow Structure

- **States:**
  1. (a) code of MPI call, or
  2. (b) code between MPI calls

- **Edges:**
  1. Transition probability
  2. Time distribution

Find offline:

- phase
- task
- code region

**Semi-Markov Models (SMM)**

- **States:**
  - $S_1$
  - $S_2$
  - $S_3$

- **Edges:**
  - $S_1$ to $S_1$, $S_2$ to $S_2$, $S_3$ to $S_3$

Example:

- Send() after Send with probability 0.6
Detection of Anomalous Phase / Task / Code-Region

- Dissimilarity between models $Diss (SMM_1, SMM_2) \geq 0$
- Cluster the models
  - Find unusual cluster(s)
  - Use known ‘normal’ clustering configuration

Find offline:
- phase → task → code region

Master-slave program

Unusual cluster
Faulty tasks
Making AutomaDeD scalable: What was necessary?

1. Replace offline clustering with fast online clustering
   • Offline clustering is slow
   • Requires data to be aggregated to a single node
AutomaDeD uses CAPEK for scalable clustering

- CAPEK algorithm scales logarithmically with number of processors in the system
  - Runs in less than half a second at with 131k cores
  - Feasible to cluster on trans-petascale and exascale systems
- CAPEK uses aggressive sampling techniques
  - This induces error in results, but error is 4-7%
Making AutomaDeD scalable: What was necessary?

1. Replace offline clustering with fast online clustering

2. Scalable outlier detection
   - By nature, sampling is unlikely to include outliers in the results
   - Need something to compensate for this
   - Can’t rely on the “small” cluster anymore
We have developed scalable outlier detection techniques using CAPEK

**Clustering Approach**

1. Perform clustering using CAPEK
2. Find distance from each task to its medoid
3. Normalize distances using within-cluster standard deviation
4. Find top-k outliers sorting tasks based on the largest distances

- The algorithm is fully distributed
- Doesn’t require a central component to perform the analysis
- Complexity $O(\log \#\text{tasks})$
Making AutomaDeD scalable: What was necessary?

1. Replace offline clustering with fast online clustering
2. Scalable outlier detection
3. Replace glibc backtrace with LLNL callpath library
   - Glibc backtrace functions are slow, require parsing
   - Callpath library uses module-offset representation
     - Can transfer canonical callpaths at runtime
     - Can do fast callpath comparison (pointer compare)
Making **AutomaDeD** scalable: What was necessary?

1. Replace offline clustering with fast online clustering
2. Scalable outlier detection
3. Replace glibc backtrace with LLNL callpath library
   - Glibc backtrace functions are slow, require parsing
   - Callpath library uses module-offset representation
     - Can transfer canonical callpaths at runtime
     - Can do fast callpath comparison (pointer compare)
4. Graph compression for Markov Models
   - MM’s contain a lot of noise
   - Noisy models can slow comparison and obfuscate problems
Too Many Graph Edges: The Curse of Dimensionality

Sample SMM graph of NAS BT run

~ 200 edges in total

- Too many edges = Too many dimensions
- Distances between points tend to become almost equal
- Poor performance of clustering analysis
Example of Graph Compression

Sample code

```c
MPI_Init() // comp code 1
MPI_Gather() // comp code 2
for (...) {
    // comp code 3
    MPI_Send() // comp code 4
    MPI_Recv() // comp code 5
}
// comp code 6
MPI_Bcast() // comp code 7
MPI_Finalize()
```

Semi-Markov Model

```
Init
  ↓
Comp 1
  ↓
Gather
  ↓
Comp 2,3
  ↓
Send
  ↓
Comp 4
  ↓
Recv
  ↓
Comp 5,6,3
  ↓
Bcast
  ↓
Comp 7
  ↓
Finalize
```

Compressed Semi-Markov Model

```
Init
  ↓
Send
  ↓
Comp 5,6,3
  ↓
Finalize
```

Sample code

```
Semi-Markov Model = Compressed Semi-Markov Model
```
We are developing AutomaDeD into a framework for many types of distributed analysis

Our graph compression and scalable outlier detection enables automatic bug isolation in:

- ~ 6 seconds with 6,000 tasks on Intel hardware
- ~ 18 seconds at 103,000 cores on BG/P

Logarithmic scaling implies billions of tasks will still take less than 10 seconds

We are developing new on-node performance models to target resilience problems as well as debugging.
We have added probabilistic root cause diagnosis to AutomaDeD

- Scalable AutomaDeD can tell us:
  - Anomalous processes
  - Anomalous transitions in the MM

- We want to know what code is likely to have caused the problem
  - Need more sophisticated analysis for this
  - Need distributed dependence information to understand distributed hangs
Progress Dependence

- Very simple notion of dependence through MPI operations
- Support collectives and point to point
  - MPI_Isend, Irecv running dependence through completion operation (MPI_Wait, Waitall, etc)
- Similar to postdominance, but there is no requirement that there be an exit node
  - Exit nodes may not be there in the dynamic call tree, especially if there is a fault

Sample code

10 // Computation code ...
11 MPI_Bcast(..., MPI_COMM_WORLD);
12 // ...
13 if (...) {
14   // ...
15   MPI_Reduce(..., comm_1);
16   // ...
17   MPI_Barrier(comm_1);
18 } else {
19   // ...
20   MPI_Bcast(..., comm_2);
21 }
22 // ...

Progress dependence graph

- Task a
- Task group B
- Task group C
- Task group D
- Task group E

Computation code

- Bcast
- Reduce
- Barrier
Building the Progress Dependence Graph (PDG)

- We build the PDG using a parallel reduction
- We estimate dependence edges between tasks using control flow stats from our Markov Models
- PDG allows us to find the least-progress (LP) process

<table>
<thead>
<tr>
<th>No</th>
<th>Task x</th>
<th>Task y</th>
<th>Union</th>
<th>Reasoning</th>
<th>OR operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$i \rightarrow j$</td>
<td>null</td>
<td>$i \rightarrow j$</td>
<td>first dependence dominates</td>
<td>$1 + 0 = 1$</td>
</tr>
<tr>
<td>2</td>
<td>$i \rightarrow j$</td>
<td>$i \rightarrow j$</td>
<td>$i \rightarrow j$</td>
<td>same dependence</td>
<td>$1 + 1 = 1$</td>
</tr>
<tr>
<td>3</td>
<td>$i \leftarrow j$</td>
<td>$i \leftarrow j$</td>
<td>$i \leftarrow j$</td>
<td>same dependence</td>
<td>$2 + 2 = 2$</td>
</tr>
<tr>
<td>4</td>
<td>$i \rightarrow j$</td>
<td>$i \leftarrow j$</td>
<td>$i?j$</td>
<td>undefined</td>
<td>$1 + 2 = 3$</td>
</tr>
<tr>
<td>5</td>
<td>null</td>
<td>null</td>
<td>null</td>
<td>no dependence</td>
<td>$0 + 0 = 0$</td>
</tr>
</tbody>
</table>

Table 2: Some examples of dependence unions.
Our distributed pipeline enables fast root cause analysis

- Full process has $O(\log(P))$ complexity
- Distributed analysis requires < 0.5 sec on 32,768 processes
- Gives programmers insight into the exact lines that could have caused a hang.
- We use DynInst’s backward slicing at the root to find likely causes
We have used AutomaDeD to find performance problems in a real code

- Discovered cause of elusive hang in I/O phase of ddcMD molecular dynamics simulation at scale
- Only occurred with 7,000+ processes